CS422 - Final Exam

Time: you have 48 hours, from the time you receive it. Send it either by email or push it under my door (SC 2110).
Total: 100 points. You are *not* allowed to collaborate or discuss these problems with each other. You can use books, lecture notes, computers, coffee, etc.

Problem 1. (15 points)
Consider programs of the following form, written in a `FUN`-like language supporting the various parameter passing styles discussed in class:

```
let f(x{P}, y{P}) = x * y
and g(x{Q}, y{Q}) = (
                        & x := x + y ;
                        & y := x - y ;
                        & x := x - y ;
                        x + y
                    )
and x = a and y = b
in f(g(x,y), g(y,y))
```

Here `a` and `b` are some arbitrary integers, assumed given. `P` and `Q` stay for parameter passing styles. What values do these programs evaluate to, when `P` and `Q` are any of call-by-value, call-by-name, or call-by-need? For each of the nine situations, explain how you obtained the result.

Problem 2. (20 points)
Suppose that you are the designer of a concurrent language and that you'd like to add support for transactional actions into your language by means of an atomicity construct. More precisely, you'd like to add a statement atomic{E}, which evaluates E atomically, that is, without any interference from the other threads (assume that you have already added thread creation and termination to your language). Also, you'd like to add a statement break-atomic, which leaves the atomic block discarding all the computation together with its side effects that have been accumulated while partially executing the atomic block, and unfreezes the rest of the threads. If the atomic block terminates normally, then its side-effects are committed. In other words, atomic{E} freezes the other threads and starts to evaluate E optimistically; if E evaluates normally, then its effects are committed, the other threads are unfrozen, and the computation continues normally; if the evaluation of E encounters a break-atomic statement, then the state of the program is recovered to the moment in which the atomic block was tried as if none of its code has been executed, the atomic block is skipped, the other threads are unfrozen, and the computation continues normally, as if the atomic block was never seen.

This problem has two parts:

1. Sketch K definitions for atomic{E} and break-atomic;
2. Comment on the computational limitations of your definition above, namely on the fact that the rest of the world is frozen in order for an atomic block to stay atomic. Can you do better?

Can you let the other threads continue their executions and stop them only if they try to interfere with the atomic block?

Problem 3. (20 points)
Suppose that you have the following object-oriented program in a language combining OO and functional features, where `initialize` is the implicit constructor method of a class:

```
class A {
  field value;
  method initialize(v) { value := v }
  method m() { value + 10 }
}
class B inherits A {
  method m() { super m() }
}
class C inherits B {
  field obj;
  method void initialize(v,o) {
    super initialize(v);
    obj := o
  }
  method m() { value + send obj m() }
}
class D inherits B {
  method initialize(v) { super initialize(v) }
  method m() { value * 2 }
}
main
  let b = ...
  in let o = if b then new C(5, new C(10, new D(15))) else new D(20)
     in send o m()
```

Translate and run this program in SKOOL. What value will be returned under dynamic method invocation when b is true? When b is false? Recall that SKOOL is a dynamically dispatched language, like Java. Under static method dispatch, one would statically analyze the program and find precisely which method is being invoked by each call. The main advantage of static dispatch is its efficiency: one needs no runtime search. C++ has static method dispatch by default, but also provides support for dynamic method dispatch (using virtual methods). Comment on what you'd need to do to transform SKOOL into a statically dispatched language. Would you need to take typing into account? How about conditionals? What values would the program above evaluate to under static method dispatch, when b is true and when b is false?

Problem 4. (25 points)

(10 points) Write FUN implementations of the classic higher-order functions `map` (for example, `map f [a,b,c,...] = [f(a),f(b),f(c),...]`) and `fold` (for example, `fold f x [a,b,c,...]`

`= ... (f (f (f x a) b) c) ...`). Write a FUN program for each of the above, of the form "`letrec function ... in function`".

(15 points) Type the programs above by hand using the polymorphic type inference technique discussed in class. Give enough detail to show that you understand the technique, but not more than that.

Problem 5. (20 points)

Parallel assignments, typically written `X1, X2, ..., Xn := E1, E2, ..., En`, can be a useful feature of a programming language, because they allow one to write more compact, abstract and intuitive code. A parallel assignment first evaluates the expressions in its right-hand-side *in parallel* and then assigns the obtained values to the names listed in its left-hand-side, in the corresponding order. For example, `x,y := y,x` swaps the values of `x` and `y` without a need for a temporary variable. Suppose that you, as a language designer, want to add such parallel assignments to your programming language (it does not matter which one; for simplicity you may assume SILF or FUN, if you need to). Also, suppose that you want to take full advantage of a highly parallel rewrite engine, in the sense of evaluating the `n` expressions indeed in parallel, as if they are different execution threads. This problem has two parts:

1. Show that the parallel assignment feature is just "syntactic sugar", in the sense that it can be automatically translated into an equivalent expression using only the already existing language features;

2. Give the parallel assignment a direct continuation-based semantics using K.

3. Give small-step SOS, big-step SOS, and context reduction definitions of parallel assignment as well, and comment on their advantages (if any) and disadvantages, especially with respect to your initial goal, namely to process the involved expressions *in parallel*.

You do not need to redefine anything that was already defined in the lecture notes.